

Practical State Machine Design Using VHDL

By Sundar Rajan*

Like ancient Gaul, all state machines can be divided into three parts... which simplifies their design greatly.

*Edited with additional material added by Ross Snider for EE367 at Montana State University
adapted from <http://www.eetimes.com/editorial/1995/fpgafeature9502.html>

Designing finite state machines (FSMs, or simply “state machines”) is a common task for digital design engineers. Signal controllers, arbiters, and waveform generators are but some of the examples of sequential circuits that are typically designed using state machines. Historically, the approach for designing a state machine has been to draw a state (or “bubble”) diagram, then translate the diagram into a schematic. More recently, however, it’s become common to translate the state diagram into a hardware description language (HDL) such as Verilog, ABEL, or VHDL for logic synthesis.

Although it is a rich and powerful language, VHDL suffers in logic synthesis from a lack of uniformity among vendors. Different vendors use different subsets of VHDL, which has hindered the language’s promise of portability and design migration.

This article gives you a four-step procedure for the design of synchronous state machines in VHDL. The methodology presented uses “legal” VHDL and focuses on a style and syntax that is inherently portable and independent of technology and architecture. And it considers practical issues, such as state encoding and one-hot-encoding techniques. If you follow the methodology presented here, you should be successful in capturing your VHDL descriptions, and you will also be writing code that is readable and highly portable.

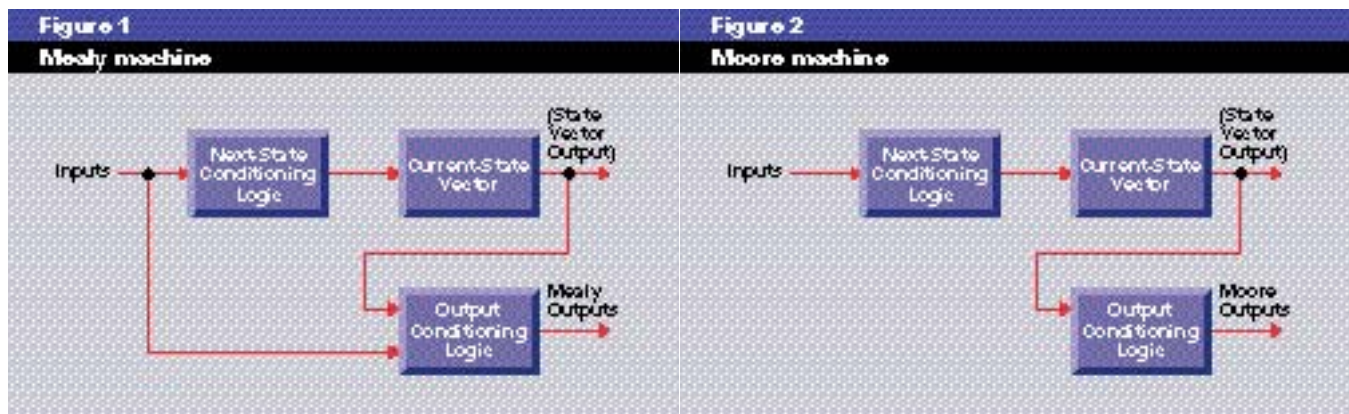
State machine types

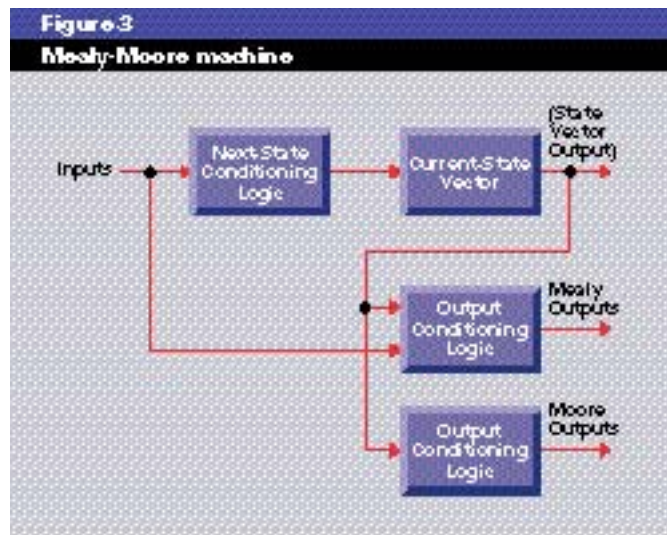
In general, all state machines can be divided into three functional blocks:

1. Next-state conditioning logic
2. State-registers (or state-variables)
3. Output conditioning logic.

This article, and you, will make use of this three-block model to describe a state machine in VHDL using our **four-step design procedure**.

Moreover, the outputs of a state machine define its type. That is, a Mealy machine is one in which the outputs are a function of both the inputs and the current state-variables (Figure 1). A Moore machine has outputs that are a function of the state-variables only (Figure 2). And a Mealy-Moore machine, as the name suggests, has both Mealy- and Moore-type outputs (Figure 3).



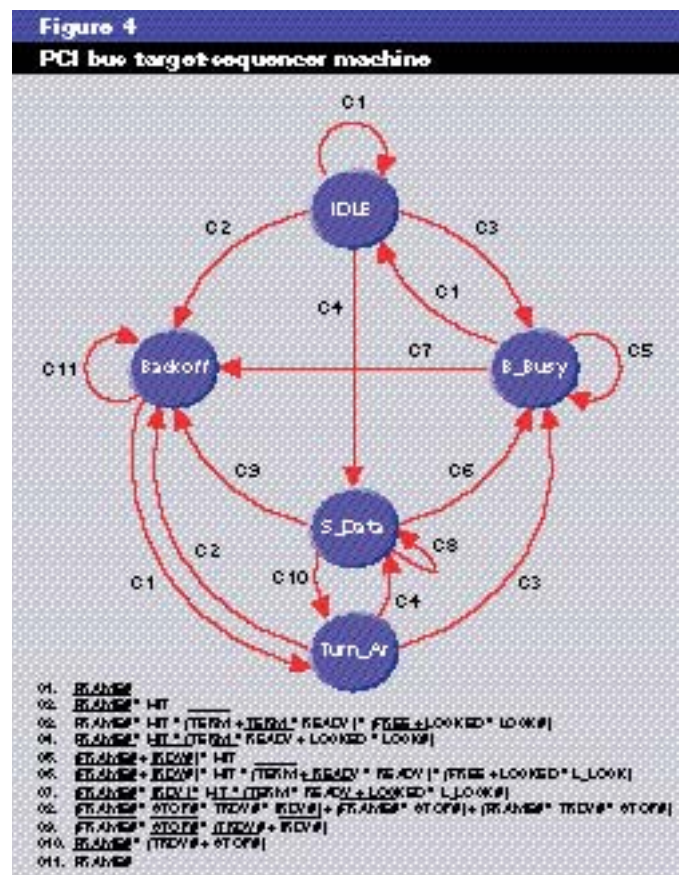


In practice, most state machines are Mealy machines, since there are usually one or more outputs that are a function of both the inputs and the current state-variables--even though you may find other outputs that are a function of the state variables only. Don't worry about it, however; VHDL's flexibility is such that you can write behavioral models without knowing a priori which type of state machine (Mealy, Moore, or Mealy-Moore) you are describing. This is because VHDL processes (which define specific behaviors) run concurrently and you will separate the output-conditioning logic into its own process so that the type of output (Mealy or Moore) is determined automatically.

In its essence, then, VHDL coding methodology is independent of the type of machine being developed. Hence the following material, particularly the coding style presented, is also independent of the type of machine being developed. We will therefore look at a state machine with both Mealy and Moore outputs--a state machine that is, in effect, a superset of either a Mealy-only or Moore-only machine.

Coding your design in VHDL

The **first part** of our four-step VHDL design procedure is to capture the concept of the design (its algorithm) using a state transition diagram. As a design example, this article considers a peripheral component interconnect (PCI) local bus target-sequencer machine (Figure 4 and sidebar).



The **second step** is to define the state machine's inputs and outputs. In VHDL, you treat the machine as a "black box"--that is, an entity-architecture pair for which the state machine's I/Os are the entity's ports. (The entity-architecture pair is the fundamental component of a VHDL behavioral description; the entity defines the interface of the design element to the external system, while the architecture defines the internal architecture and behavior of the entity.) For reasons of clarity, here we will code only the output-enable generated by the state machine, not all its outputs. The entity description is seen in Listing 1.

Listing 1 - Entity Description

```
entity pciTargetFSM is port (
    clk,reset: in bit;
    FRAME_L,RDY_L,TRDYi_L,
    IRDY_L,STOP_L,           -- PCI inputs
    hit,term,ready,L_lock_l, tar_dly, -- inputs from PCI back-end
    cmd_r_w: in bit;
    LockFSMFree, LockFSMLocked: in bit; -- inputs from target lock machine

    OE_AD,OE_TRDY_L,OE_STOP_L,OE_DEVSEL_L, -- outputs of FSM
    OE_PAR: out bit
);
end pciTargetFSM;
```

The **third step** requires that you define the states of the machine, the behavior (the body, or architecture) of the FSM entity just described. You begin by defining the states as an enumerated type. VHDL is a strongly typed language and provides a variety of data types; enumerated types are simply types that can be defined by the user. You can use enumerated types to define all the states in the machine.

```
type TargetSeqStates is (IDLE,B_BUSY,BACKOFF,S_DATA,TURN_AR);
signal curr_st,next_st: TargetSeqStates;
```

The **fourth and final step** is to write the body of the state machine. We have already noted the natural breakdown of a state machine into three parts. This allows you to divide a machine's VHDL description into three VHDL processes. Think of a process (i.e. a behavior) as a collection of sequential statements, and remember that processes execute simultaneously (in parallel). Note, too, that each of the processes you will write is equivalent to one of the three parts of the machine. **Thus, the way to capture a state machine in VHDL is to (1) write a combinatorial process that performs the next-state conditioning logic, (2) write a synchronous process that creates the current-state variables, and, (3) add the process that creates the output conditioning logic.**

Condition the next state

As just noted, **the first thing to do is to write a combinatorial process that performs the next-state conditioning logic**. Per VHDL syntax, this process is sensitized to all the inputs, as well as the current state. This is because the next state to which the state machine transitions is a function of the current state and the inputs. For this process, use the case statement. In VHDL, the case statement is used to select one of a series of choices based on the state of a signal. Here, the state of the signal is the current-state of the machine. (See Listing 2.)

Listing 2 - The case statement is used to select one of a series of signals

```
NxtStDecode: process (curr_st,FRAME_L,RDY_L,TRDYi_L,IRDY_L,STOP_L, hit, ready,
term,L_lock_1,LockFSMFree,LockFSMLocked)
```

```
begin
```

```
    next_st <= IDLE; -- default next-state condition
```

```
    case curr_st is
```

```
        when IDLE | TURN_AR =>
```

```
            if FRAME_L = '0' then
```

```
                next_st <= IDLE;
```

```
            else
```

```
                if (hit = '0') then
```

```
                    next_st <= B_BUSY;
```

```
                else
```

```
                    if ((term = '0' or (term = '1' and ready = '1')) and (LockF
SMFree = '1' or (LockFSMLocked = '1' and L_lock_1 =
'0')) then
```

```
                        next_st <= S_DATA;
```

```
                    end if;
```

```
                    if (((term = '1' and ready = '0') or (LockFSMLocked = '1'
and L_lock_1 = '1')) then
```

```
                        next_st <= BACKOFF;
```

```
                    end if;
```

```
                end if;
```

```
            end if;
```

Listing 2 Continued

```
when B_BUSY =>
    if FRAME_L = '0' then
        next_st <= IDLE;
    else
        if (hit = '0') then
            if (FRAME_L = '1' or IRDY_L = '1') then
                next_st <= B_BUSY;
            end if;
        else
            if (term = '0' OR (term = '1' and ready = '1')) and (LockFSMFree
            = '1' or (LockFSMLocked = '1' and L_lock_1 = '0')) then
                next_st <= S_DATA;
            end if;
            if (FRAME_L = '1' or IRDY_L = '1') and hit = '1' and ((term =
            '1' and ready = '0') or (LockFSMLocked = '1' and L_lock_1 =
            '1')) then
                next_st <= BACKOFF;
            end if;
        end if;
    end if;

when S_DATA =>
    if (FRAME_L = '1' and STOP_L = '1' and TRDYi_L = '1' and IRDY_L = '1')
    or (FRAME_L = '1' and STOP_L = '0') or (FRAME_L = '0' and TRDYi_L = '0'
    and IRDY_L = '0') then
        next_st <= S_DATA;
    end if;
    if (FRAME_L = '1' and STOP_L = '1' and (TRDYi_L = '0' or IRDY_L = '1'))
    then
        next_st <= BACKOFF;
    end if;
    if (FRAME_L = '0' and (TRDYi_L = '1' or STOP_L = '1')) then
        next_st <= TURN_AR;
    end if;

when BACKOFF =>
    if FRAME_L = '1' then
        next_st <= BACKOFF;
    else
        next_st <= TURN_AR;
    end if;

when others =>
    next_st <= IDLE;

end case;
end process;
```

Create the current-state variables

The second step is to write a synchronous process that creates the current-state variables. The process will have two parts: one to reset the state machine asynchronously to a known state (the idle state), and the other to register the current-state condition on the rising edge of the state machine's clock.

```
RegStVector: process (clk,reset)
begin
    if (reset = '1') then
        curr_st <= IDLE;
    elsif (clk'event and clk = '1') then
        curr_st <= next_st;
    end if;
end process;
```

Condition the outputs

Finally, add the process that creates the output conditioning logic. This is shown in Listing 3.

Listing 3 - Output Conditioning

```
DecOutputs: process (curr_st)
begin
    if (curr_st = S_DATA and tar_dly = '1' and cmd_r_w = '1') then
        OE_AD <= '1';
    else
        OE_AD <= '0';
    end if;
    if (curr_st = BACKOFF or curr_st = S_DATA or curr_st = TURN_AR) then
        OE_TRDY_L <= '1';
    else
        OE_TRDY_L <= '0';
    end if;
    if (curr_st = BACKOFF or curr_st = S_DATA or curr_st = TURN_AR) then
        OE_STOP_L <= '1';
    else
        OE_STOP_L <= '0';
    end if;
    if (curr_st = BACKOFF or curr_st = S_DATA or curr_st = TURN_AR) then
        OE_DEVSEL_L <= '1';
    else
        OE_DEVSEL_L <= '0';
    end if;
    if (curr_st = S_DATA and TRDYi_L = '1' and cmd_r_w = '1') then
        OE_PAR <= '1';
    else
        OE_PAR <= '0';
    end if;
end process;
```

State-variable initialization

In the code fragments of the previous sections, there are three key points to note:

1. The initialization of the next state variable
2. The usage of the case statement
3. The usage of the if...then statement.

Initialization of the state variable allows you to set a default state for the state machine. Note that, in VHDL, the value of a signal does not get updated until the process has finished executing, at which time the signal takes on the last value that it was assigned. Thus, for our example machine, while the initial value may be the default, or idle, state, by stepping sequentially through the case statement it will reach the current state, at which point it determines the state to which it should transition.

Case statement usage

There are two important points to remember when using the case statement. The **first** is that VHDL requires you to specify completely all case possibilities (that is, the case statement must define “when” conditions for each value that the signal can take). A good VHDL analyzer will flag incompletely defined case statements, which is very useful as a debugging tool.

The **second point** is that you should close each case statement with a *when others* clause. While this is not absolutely necessary if you have completely specified the case statement, it is good design practice. It does no harm to your design, and it will help if you make a change to your state machine after you have defined the enumerated state type. For example, if you make a change to your machine such that several states are eliminated, the when others clause will take care of the unspecified states, so you don’t need to redefine the enumerated state type.

If...then statement usage

There is just one guideline to follow here, but it will affect the quality of your synthesized code significantly. Make sure you close all if...then statements in your design with an else statement. Signals in VHDL have “implicit memory,” so if they are not assigned, they “remember” their last values, and this “remembering” demands a memory element of some sort. Some design tools synthesize a latch memory element to ensure that the synthesized result exhibits exactly the same behavior as the source VHDL. Other VHDL synthesis tools are not quite so true to this aspect of the VHDL standard. Thus, particularly for reasons of portability, **it is very important to close all if...then statements with an else statement**. As an added benefit, it will make your code much more readable, particularly for someone trying to understand your design many months after you coded it.

Synthesis considerations

The methodology outlined so far will certainly work and is a good place to start. There are, however, certain realities you will face in getting your design to work and simultaneously dealing with synthesis tools. As one example, the enumerated type defined above does not imply any state encoding specification. Thus, the synthesis tool will require a certain amount of intelligence, so that it can pick the best state-encoding scheme, one that results in the most-efficient circuit realization. What do you do if your design tool doesn't perform well on this task? Well, in the following sections we identify some specific issues that arise during state machine design, show why you may want to move away from the design template specified above, and propose solutions for solving these real-world issues.

Specifying the state encoding

There will be times when you wish to specify the encoding scheme for the state machine you're designing. Some design tools take advantage of so-called "user-defined" VHDL attributes.

If, however, design portability is of greater importance than the convenience of a single attribute, specify the state encoding using VHDL constants. You do this by specifying the width of the state-variable using VHDL's `bit_vector` type, and then defining each state variable as a constant with the appropriate encoding. For example:

```
subtype TargetSeqStates is bit_vector(3 downto 0);
signal curr_st,next_st: TargetSeqStates;
constant IDLE      : TargetSeqStates := "001";
constant B_BUSY    : TargetSeqStates := "100";
constant BACKOFF   : TargetSeqStates := "101";
constant S_DATA     : TargetSeqStates := "110";
constant TURN_AR   : TargetSeqStates := "010";
```

This simply replaces the line that defines the state type as an enumerated type and adds the constants that define the state vector.

Specifying one-hot encoding

One-hot encoding (also called "bit-per-state encoding") has become popular with the advent of register-rich, finer-grain FPGA architectures. In such architectures, an encoded form of state assignment results in degraded performance, due to the additional logic cells used to encode and decode the states.

In the one-hot-encoding approach, however, only one bit in the state vector is active, or "hot," for any given state. This greatly simplifies state decoding, since you only have to examine a single bit to determine the state of the machine. Thus, the one-hot-encoded approach not only offers better performance, but often uses fewer logic cells, since the state decoding is far simpler. As indicated above, some tools allow one-hot state encoding using a single VHDL attribute. However, writing code that is portable is a little more involved than the fully specified state-encoding example. For a partial description of the state machine that uses an explicit one-hot-encoding style, see Listing 4.

Listing 4 - An explicit one-hot encoding style

```
subtype TargetSeqStates is bit_vector(4 downto 0);
signal curr_st,next_st: TargetSeqStates;
constant DEFAULT_STATE : TargetSeqStates := "00000";
constant IDLE           : integer := 0;
constant B_BUSY         : integer := 1;
constant BACKOFF        : integer := 2;
constant S_DATA         : integer := 3;
constant TURN_AR        : integer := 4;

NxtStDecode: process (curr_st,FRAME_L,RDY_L,TRDYi_L,IRDY_L,STOP_L, hit, ready,
term, L_lock_l,LockFSMFree,LockFSMLocked)

begin
    next_st <= IDLE; -- default next state condition

    if curr_st(IDLE) = '1' then
        if FRAME_L = '0' then
            next_st(IDLE) <= '1';
        else
            if (hit = '0') then
                next_st(B_BUSY) <= '1';
            else
                if (term = '0' or (term = '1' and ready = '1')) and (LockFSMFree =
'1' or (LockFSMLocked = '1' and L_lock_l = '0')) then
                    next_st(S_DATA) <= '1';
                end if;
                if ((term = '1' and ready = '0') or (LockFSMLocked = '1' and
L_lock_l = '1')) then
                    next_st(BACK_OFF) <= '1';
                end if;
            end if;
        end if;
    else
        next_st(IDLE) <= '1';
    end if;

    .... etc. ...
```

What you should note in the partial listing 4 is the use of a series of *if...then* statements rather than the *case* statement. There is a one-to-one correspondence between each when condition in a *case* statement and each *if* statement, except when specifically activating a single bit in the state vector representing the state being transitioned to. Note that all other state bits will be zero, since the first assignment made in the process zeroes all the next-state bits:

```
next_st <= IDLE; -- default next state condition
```

Output optimization

When working with programmable logic, you've probably used "don't-care" specifications in your code to simplify the logic. If you can guarantee, by design, that certain outputs will never be sampled during certain states, you can take advantage of such situations to allow the synthesis tool some flexibility during logic optimization. For instance, a don't-care specification on a particular output might let the synthesis tool eliminate redundancies, resulting in a simpler, more efficient circuit.

If you wish to specify don't-care conditions for one or more outputs, the first thing to do is to change the output type. As has already been indicated, VHDL is a strongly typed language, and the elements of `bit_vector` types can be assigned only two values, 0 and 1. So change the type you are using to a type (defined by IEEE Std. 1164) called `std_ulogic_vector`. It's similar to a `bit_vector`, but can be specifically assigned a don't-care value. Thus, your new state machine description might look like Listing 5.

Listing 5 - Specifying "don't-care" conditions

```
OE_AD,OE_TRDY_L,OE_STOP_L,OE_DEVSEL_L, -- outputs of FSM
```

```
OE_PAR: out std_ulogic;
```

```
DecOutputs: process (curr_st)
begin
```

```
    if (curr_st = S_DATA and tar_dly = '1' and cmd_r_w = '1') then
        OE_AD <= '1';
    else
        OE_AD <= '-';
    end if;
```

```
    ...
    ...
```

```
end process;
```

Note that the basic three-process design methodology has not really changed. Some minor modifications to the type and to the output-conditioning logic process have been made, but the core concept remains the same. (Also, the example noted is strictly for the purpose of illustrating a point. It is not called out in the PCI 2.0 specification.)

Design verification

It's beyond the scope of this article, but design verification should be mentioned here. If you simulate the source VHDL, you will be able to debug your design quite easily. Note, however, that, during the synthesis process, enumerated types will eventually get converted to registers for the state vector. Also, `bit_vectors` will get converted to "bits" or "wires." For verification purposes, then, make sure you understand the state encoding chosen by the tool, so that you can perform a post-synthesis simulation.

Summing up

VHDL is a rich, powerful hardware description language that permits many different levels of design abstraction and several ways to articulate a circuit. A result of such flexibility is that there are various "legal" ways to describe a design, but the various ways may be neither very readable nor portable across different tools.

To eliminate such problems, this article has presented a simple methodology for designing state machines in VHDL. The methodology separates a state machine into three parts--the next-state conditioning logic, the state variable, and the output-conditioning logic--which are easily coded as three VHDL processes. It has also demonstrated that specific issues the designer wishes to address--such as specific state encoding, one-hot encoding, or don't-care output assignments--are easily handled by slightly modifying the given methodology. While there are several ways to achieve the same results using other coding styles and methodologies, the ones presented here focus specifically on the VHDL code's readability and portability.

*Sundar Rajan is a Cypress staff engineer. He is a project leader in the applications group with specific responsibilities involving architectures for programmable logic.
integrated system design February 1995*

Additional Note:

You can take advantage of the fact that in a process, only the last assignment holds true before the process waits at the bottom of the process. The process only runs again from top to bottom when there is a signal change for any signal listed in the sensitivity list.

Suppose you want LED5 to be lit when the state machine is only in state 5. Rather than specifying the LED5 value in each state, you can create a default value that sets LED5 at the beginning of the process, and then only specify the change when the FSM is in state 5.

```
process(state)
begin

    LED5 <= '0'; -- LED5 is set to zero as a default value

    case state is
        when S0 =>
        when S1 =>
        when S2 =>
        when S3 =>
        when S4 =>
        when S5 =>
            LED5 <= '1'; -- LED5 will only change to 1 (lit) when state = S5
        when S6 =>
        when others =>
    end case;
end process;
```

The above coding style is easier to code and easier to read.

rks: additional coding style.

A State Machine Template

```
-- Example of Mealy/Moore state machine
--      clk : in std_logic;
--      reset : in std_logic;
--      inp1, inp2 : in std_logic;
--      outp : out std_logic

--Insert the following statements in the architecture declarative part:
type state_type is (state_1, state_2, state_3);
signal state, next_state : state_type;

--Declare internal signals for outputs
signal outp_int : std_logic;

--other outputs...

begin

--Insert the following statements in the architecture body:
--This process transitions to the next state on the rising edge of the clock
process (clk)
begin
    if clk'event and clk = '1' then
        if reset = '1' then
            state <= state_1; -- reset state
        else
            state <= next_state; -- transition to next state
        end if;
    end if;
end process;

--Mealy or Moore State Machine depending on whether the output is a combination
--of both the current state and inputs (Mealy) or just the current state (Moore).
decode_outp: process (state, inp1)
begin
    --insert statements to decode internal output signals
    --below is simple example
    if state = state_3 and inp1 = '1' then
        outp_int <= '1';
    else
        outp_int <= '0';
    end if;
end process;

--Note: A Moore Machine can use a case statement to select the state.
```

```

-- This process decides what the next state is based on inputs
decode_next_state: process (state, inp1, inp2)
begin
    --declare default state for next_state to avoid latches
    next_state <= state; --default is to stay in current state

    --insert statements to decode next_state
    case (state) is
        when state_1 =>
            if inp1 = '1' then
                next_state <= state_2;
            end if;
        when state_2 =>
            if inp2 = '1' then
                next_state <= state_3;
            end if;
        when state_3 =>
            next_state <= state_1;
        when others =>
            next_state <= state_1;
    end case;
end process;

```

rks: adapted from Aldec's Active-HDL synthesis template.